Consistency between Task Models and Use Cases

Daniel Sinnig[1], Patrice Chalin[1] and Ferhat Khendek[2]

[1] Department of Software Engineering and Computer Science,
Concordia University, Montreal, Quebec, Canada
{d_sinnig, chalin}@encs.concordia.ca

[1] Department of Electrical and Computer Engineering,
Concordia University, Montreal, Quebec, Canada
khendek@ece.concordia.ca

Abstract. Use cases are the notation of choice for functional requirements documentation, whereas task models are used as a starting point for user interface design. In this paper, we motivate the need for an integrated development methodology in order to narrow the conceptual gap between software engineering and user interface design. This methodology rests upon a common semantic framework for developing and handling use cases and task models. Based on the intrinsic characteristic of both models we define a common formal semantics and provide a formal definition of consistency between task models and use cases. The semantic mapping and the application of the proposed consistency definition are supported by an illustrative example.

Keywords: use cases, task models, finite state machines, formal semantics, consistency

1 Introduction

Current methodologies and processes for functional requirements specification and UI design are poorly integrated. The respective artifacts are created independently of each other. A unique process allowing for UI design to follow as a logical progression from functional requirements specification does not exist. Moreover, it has been noted that most UI design methods are not well integrated with standard software engineering practices. In fact, UI design and the engineering of functional requirements are often carried out by different teams using different processes [1].

There is a relatively large conceptual gap between software engineering and UI development. Both disciplines have and manipulate their own models and theories, and use different lifecycles. The following issues result directly from this lack of integration:

- Developing UI-related models and software engineering models independently neglects existing overlaps, which may lead to redundancies and increase the maintenance overhead.

- Deriving the implementation from UI-related models and software engineering models towards the end of the lifecycle is problematic as both processes do not have the same reference specification and thus may result in inconsistent designs.

Use cases are the artifacts of choice for the purpose of functional requirements documentation [2] while UI design typically starts with the identification of user tasks, and context requirements [3]. Our primary research goal is to define an integrated methodology for the development of use case and task model specifications, where the latter follows as a logical progression from the former. Figure 1 illustrates the main component of this initiative, which is the definition of a formal framework for handling use cases and task models at the requirements and design levels. The cornerstone for such a formal framework is a common semantic model for both notations. This semantic model will serve as a reference for tool support and will be the basis for the definition of a consistency relation between a use case specification and a task model specification. The latter is the focus of this paper.

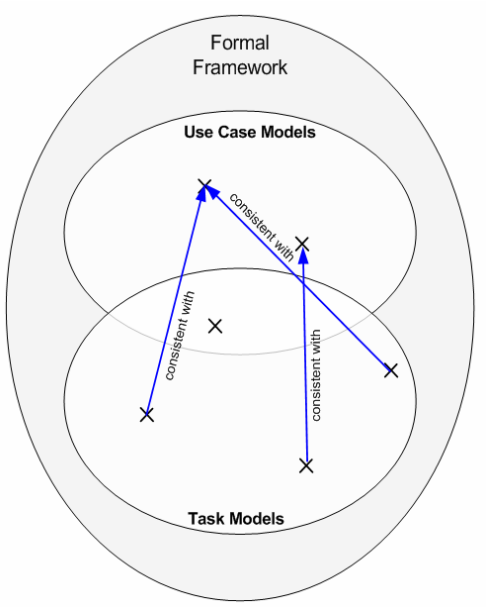


Fig. 1. Relating Use Cases and Task Models within a Formal Framework

The structure of this paper is as follows. Section 2 reviews and compares key characteristics of use cases and task models. Section 3 presents a formal mapping from use cases and task models to (nondeterministic) state machines. Based on the intrinsic characteristics of use cases and task models, we provide a formal definition of consistency. Our definition is illustrated with an example as well as with a counterexample. Finally in Section 4, we draw the conclusion and provide an outlook to future research.

2 Background

In this section we remind the reader of the key characteristics of use cases and task models. For each notation we provide definitions, an illustrative example as well as a formal representation. Finally, both notations are compared and the main commonalities and differences are contrasted.

2.1 Use Cases

A use case captures the interaction between actors and the system under development. It is organized as a collection of related success and failure scenarios that are all bound to the same goal of the primary actor [4]. Use cases are typically employed as a specification technique for capturing functional requirements. They document the majority of software and system requirements and as such, serve as a contract (of the envisioned system behavior) between stakeholders [2]. In current practice, use cases are promoted as structured textual constructs written in prose language. While the use of narrative languages makes use case modeling an attractive tool to facilitate communication among stakeholders, prose language is well known to be prone to ambiguities and leaves little room for advanced tool support.

As a concrete example, Figure 2 presents a sub-function level use case for a "Login" function. We will be using the same example throughout this paper, and for the sake of simplicity, have kept the complexity of the use case to a minimum. A use case starts with a header section containing various properties of the use case. The core part of a use case is its main success scenario, which follows immediately after the header. It indicates the most common ways in which the primary actor can reach his/her goal by using the system. The main success scenario consists of a set of steps as well as (optional) control constructs such as choice points. We note that technically and counter-intuitively to its name, the main success scenario does not specify a single scenario but a set of scenarios. However, current practice in use case writing suggests the annotation of the main success scenario with such control constructs [2]. Within our approach we acknowledge this "custom" by allowing control structures to be included in the main success scenario.

A use case is completed by specifying the use case extensions. These extensions constitute alternative scenarios which may or may not lead to the fulfillment of the use case goal. They represent exceptional and alternative behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition (relative to one or more steps of the main success scenario), which makes the extension relevant and causes the main scenario to "branch" to the alternative scenario. The condition is followed by a sequence of action steps, which may lead to the fulfillment or the abandonment of the use case goal and/or further extensions. From a requirements point of view, exhaustive modeling of use case extensions is an effective requirements elicitation device.

Use Case: Login

Goal: Customer logs into the program
Level: Sub-function
Primary Actor: Customer

Main scenario

1. Customer indicates that he/she wishes to log-in to the system. (step:interaction)
2. Customer performs the choice of the following: (stepChoice)
   2.1a Customer provides the user name. (step:interaction)
   2.1b Customer provides the password. (step:interaction)
   OR
   2.2a Customer provides the password. (step:interaction)
   2.2b Customer provides the user name. (step:interaction
3. Customer confirms the provided data (step:interaction)
4. System authenticates customer. (step:internal))
5. System informs the customer that the Login was successful. (step:interaction)
6. System grants access to customer based on his/her access levels. (step:internal)
7. *The use case ends.* (stepEnd)

Extensions

4a. The provided username or/and password is/are invalid:

4a1. The system informs the customer that the provided username and/or password is/are invalid. (step:interaction)
4a2. The system denies access to the customer. (step:internal)
4a2. *The use case ends unsuccessfully.* (stepEnd)

Fig. 2. Textual Presentation of the "Login" Use Case

As mentioned before use cases are typically presented as narrative, informal constructs. A formal mapping from their informal presentation syntax to a semantic model is not possible. Hence, as a prerequisite, for the definition of formal semantics and consistency, we require use cases to have a formal structure, which is independent of any presentation. We have developed a XML Schema (depicted in Figure 3) which acts as a meta model for use cases. As such, it identifies the most important use case elements, defines associated mark-up and specifies existing containment relationships among elements. We use XSLT stylesheets [5] to automatically generate a "readable" use case representation (Figure 2) from the corresponding XML model.

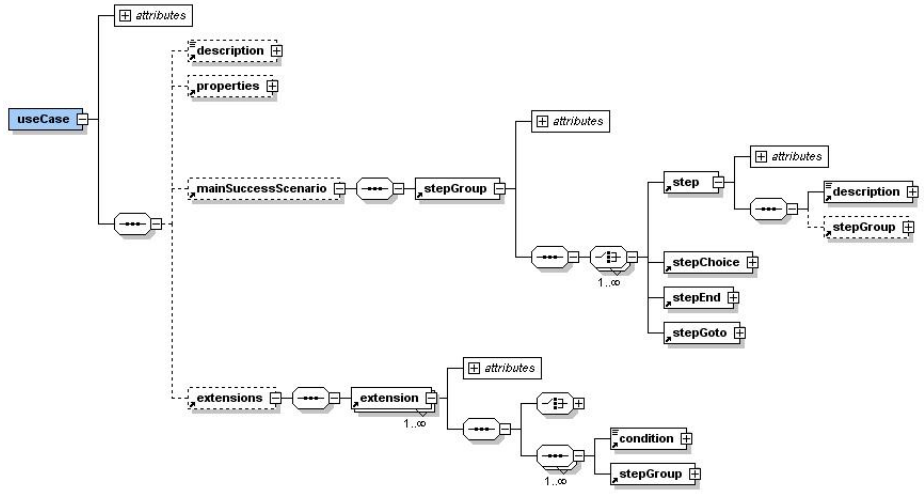


Fig. 3. Use Case Meta Model

Most relevant for this paper is the definition of the *stepGroup* element as it captures the behavioral information of the use case. As depicted, the *stepGroup* element consists of a sequence of one of the following sub elements:

- The ***step*** element denotes a use case step capturing the primary actor's interactions or system activities. It contains a textual description and may recursively nest another *stepGroup* element. As implied by the annotations in Figure 2, we distinguish between interaction steps and internal steps. The former are performed or are observable by the primary actor and require a user interface, whereas the latter are unobservable by the primary actor.
- The ***stepEnd*** element denotes an empty use case step which has neither a successor nor an extension.
- The ***stepChoice*** element denotes the alternative composition of two *stepGroup* elements.
- The ***stepGoto*** element denotes an arbitrary branching to another *step*.

We note that the *stepGroup* element is part of the *mainSuccessScenario* as well as the *extension* element. The latter additionally contains a condition and a reference to one or many steps stating *why* and *when* the extension may occur.

2.2 Task Models

User task modeling is by now a well understood technique supporting user-centered UI design [6]. In most UI development approaches, the task set is the primary input to the UI design stage. Task models describe the tasks that users perform using the application, as well as how the tasks are related to each other. Like use cases, task models describe the user's interaction with the system. The primary purpose of task models is to systematically capture the way users achieve a goal when interacting

with the system [7]. Different presentations of task models exist, ranging from narrative task descriptions, work flow diagrams, to formal hierarchical task descriptions.

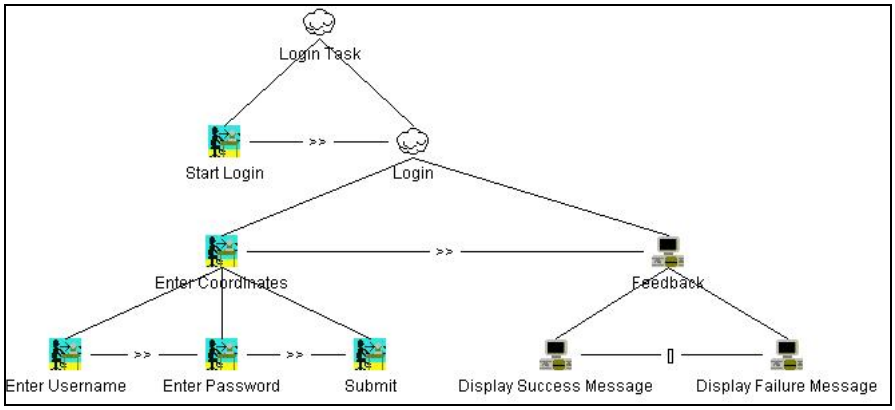


Fig. 4. "Login" Task Model

Figure 4 shows a ConcurTaskTreesEnvironment (CTTE) [8] visualization of the "Login" task model. CTTE is a tool for graphical modeling and analyzing of ConcurTaskTrees (CTT) models [9]. The figure illustrates the hierarchical break down and the temporal relationships between tasks involved in the "Login" functionality (depicted in the use case of Section 2.1). More precisely, the task model specifies how the user makes use of the system to achieve his/her goal but also indicates how the system supports the user tasks. An indication of task types is given by the used symbol to represent tasks. Task models distinguish between externally visible system tasks and interaction tasks. Internal system tasks (as they are captured in use cases) are omitted in task models.

Formally a task model is organized as a directed graph. Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. Atomic tasks are also called actions, since they are the tasks that are actually carried out by the user or the system. The execution order of tasks is determined by temporal operators that are defined between peer tasks. In CTT various temporal operators exist; examples include: enabling (>>), choice ([]), iteration (*), and disabling ([>]. A complete list of the CTT operators together with a definition of their interpretation can be found in [9].

2.3 Use Cases vs. Task Models

In the previous two sections, the main characteristics of use cases and task models were discussed. In this section, we compare both specifications and outline noteworthy differences and commonalities. In Section 3 the results of this comparison will be used as guides for the definition of a proper consistency relation that fits the particularities of both specifications.

Both use cases and task models belong to the family of scenario-based notations, and as such capture sets of usage scenarios of the system. In theory, both notations can be used to describe the same information. In practice however, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements/design details. Based on this assumption we identify three main differences which are pertinent to their purpose of application:

1. Use cases capture requirements at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of the task model are often lower level UI details that are irrelevant (actually contraindicated [2]) in the context of a use case. We note that due to its simplicity, within our example, this difference in the level of abstraction is not explicitly visible.
2. Task models concentrate on aspects that are relevant for UI design and as such, their usage scenarios are strictly depicted as input-output relations between the user and the system. Internal system interactions (i.e. involvement of secondary actors or internal computations) as specified in use cases are not captured.
3. If given the choice, a task model may only implement a subset of the scenarios specified in the use case. Task models are geared to a particular user interface and as such must obey to its limitations. E.g. a voice user interface will most likely support less functionality than a fully-fledged graphical user interface. In the next section we will address the question of which use case scenarios the task model *may* specify and which scenarios the task model *must* specify.

3 Formal Definition of Consistency

In this section we first review related work and mathematical preliminaries. Next we define the mapping from use cases and task models to the proposed semantic domain of finite state machines. Finally we provide a formal notion of consistency between use cases and task models.

3.1 Related Work

Consistency verification between two specifications has been investigated for decades and definitions have been proposed for various models [10-14]. But to our knowledge a formal notion of consistency has never been defined for use cases and task model specification.

Brinksma points out that the central question to be addressed is "*what is the class of valid implementations for a given specification?*" [15] To this effect various pre-orders for labeled transition systems have been defined. Among others the most popular ones are *trace inclusion* [16], *reduction* [15], and *extension* [12, 15, 17]. The former merely requires that every trace of the implementation is also a valid trace according to the specification. The *reduction* preorder defines an implementation as a proper reduction of a specification if it results from the latter by resolving choices that were left open in the specification [15]. In this case, the implementation may have less traces. In the case of the *extension* preorder two specifications are compared for consistency by taking into account that one specification may contain behavioral

information which is not present in the other specification. In the subsequent section we adopt (with a few modifications) the *extension* preorder as the consistency relation between uses cases and task models. A prerequisite for a formal comparison (in terms of consistency) of use cases and task models is a common semantics.

In [18] Sinnig et al. propose a common formal semantics for use cases and task models based on sets of partial order sets. Structural operational semantics for CTT task models are defined in [19]. In particular Paternò defines a set of inference rules to map CTT terms into labeled transition systems. In [20] Xu et al. suggest process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring.

In [21, 22, 23] use case graphs have been proposed to formally represent the control flow within use cases. For example Koesters et al. define a use case graph as a single rooted directed graph, where the nodes represent use case steps and the edges represent the step ordering. Leaf nodes indicate the termination of the use case [21].

In our approach we define common semantics for use cases and task model based on finite state machines. In the next section we lay the path for the subsequent sections by providing the reader with the necessary mathematical preliminaries.

3.2 Mathematical Preliminaries

We start by reiterating the definition of (non-deterministic) finite state machines (FSM) which is followed by the definitions of auxiliary functions needed by our consistency definition.

Definition 1: A **(nondeterministic) finite state machine** is defined as the following tuple: $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states.
- Σ is a finite set of symbols (the input alphabet), where each symbol represents an event.
- q_0 is the initial state with $q_0 \in Q$
- F is the set of final (accepting) states with $F \subseteq Q$
- δ: $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$ is the transition function[1], which returns for a given state and a given input symbol the set of (possible) states that can be reached.

In what follows we define a set of auxiliary functions which will be used later on for the definition of consistency between two FSMs.

Definition 2: The **extended transition function** δ^*: $Q \times \Sigma^* \rightarrow 2^Q$ is defined in a standard way as:

$$\delta^*(q_i, w) = Q_j$$

[1] λ represents the empty string. $\Sigma^0 = \{\lambda\}$

where Q_j is the set of possible states the Non-deterministic FSM may be in, having started in state q_i and after the sequence of inputs w. A formal recursive definition of the extended transition function can be found in [24].

Definition 3: The function **accept:** $Q \rightarrow 2^{\Sigma}$ denotes the set of possible symbols which may be accepted in a given state.

$$\text{accept}(q) = \{a \mid \delta^*(q, a)\}$$

Note that 'a' ambiguously denotes either a symbol or the corresponding string of one element.

Definition 4: The function **failure**: $Q \rightarrow 2^{\Sigma}$ denotes the set of possible symbols which may not be accepted (refused) in a given state. failure(p) is defined as the complement of accept (p).

$$\text{failure}(p) = \Sigma \setminus \text{accept}(p)$$

Definition 5: The **language L accepted** by a FSM $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings of event symbols for which the extended transition function yields at least one final state (after having started in the initial state q_0). Each element of L represents one possible scenario of the FSM.

$$L(M) = \{w \mid \delta^*(q_0, w) \cap F \neq \varnothing\}$$

Definition 6: The **set of all traces** generated by the NFSM $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings or sequences of events accepted by the extended transition function in the initial state.

$$\text{Traces}(M) = \{w \mid \delta^*(q_0, w)\}$$

3.3 Mapping Use Cases to Finite State Machines

In this section we define a mapping from use cases to the domain of finite state machines. It is assumed that the use case specification complies with the structure outlined in Section 2.1.

The building blocks of a use case are the various use case steps. According to the control information entailed in the use case, the various steps are gradually composed into more complex steps until the composition eventually results in the entire use case. We distinguish between sequential composition and choice composition. The former is denoted by the relative ordering of steps within the use case specification or the *stepGoto* construct, whereas the latter is denoted by the *stepChoice* element.

A use case step may have several outcomes (depending on the number of associated extensions). This has an implication on the composition of use case steps. In particular the sequential composition of two use case steps is to be defined *relative* to a given outcome of the preceding step. For example the steps of the main success

scenario are sequentially composed relative to their successful (and most common) outcome. In contrast to this, the steps entailed in use case extensions are sequentially composed relative to an alternative outcome of the corresponding "extended" steps.

Following this paradigm, we propose representing each use case step as a **finite state machine**. Figure 5 depicts a blueprint of such a state machine representing an atomic use case step. The FSM only consists of an initial state and multiple final states. The transitions from the initial state to the final states are triggered by events. Each event represents a different outcome of the step. In what follows we illustrate how the sequential composition and choice composition of use case steps are semantically mapped into the sequential composition and deterministic choice composition of FSMs.

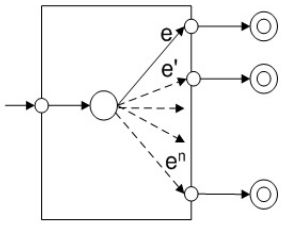


Fig. 5. FSM Blueprint for Atomic Use Case Steps

Figure 6 schematically depicts the sequential composition of two FSMs M_1 and M_2 relative to state q_n. The resulting FSM is composed by adding a transition from q_n (which is a final state in M_1) and the initial state (s_0) of M_2. As a result of the composition, both q_n and s_0 lose their status as final or initial states, respectively. The choice composition of use case steps is semantically mapped into the deterministic choice composition of the corresponding FSMs. As depicted on the left hand side of Table 1 (in Section 3.4) the main idea is to merge the initial states of the involved FSMs into one common initial state of the resulting FSM.

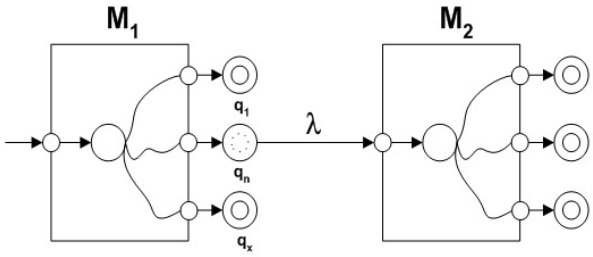


Fig. 6. Sequential Composition of Two FSMs

Figure 7 depicts the FSM representing the "Login" use case from Section 2.1. It can be easily seen how the FSM has been constructed from various FSMs representing the use case steps. Identical to the textual use case specification, the FSM

specifies the entry of the login coordinates (denoted by the events e_{21} and e_{22}) in any order. Due to the associated extension, step 4 is specified as having different outcomes. One outcome (denoted by event e_4) will lead to a successful end of the use case whereas the other outcome (denoted by event e_{4a}) will lead to login failure.

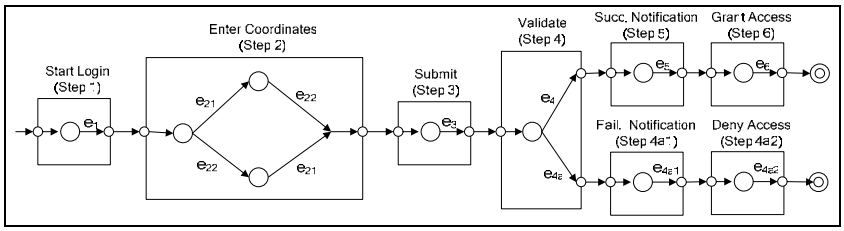


Fig. 7. FSM Representation of the "Login" Use Case

3.4 Mapping CTT Task Models to Finite State Machines

After we have demonstrated how use cases are mapped to FSM specifications, we now demonstrate the mapping from CTT task models to the same semantic domain. The building blocks of task models are the action tasks (i.e. tasks that are not further decomposed into subtasks). In CTT, action tasks are composed to complex tasks using a variety of temporal operators. In what follows we will demonstrate how actions tasks are mapped into FSMs and how CTT temporal operators are mapped into compositions of FSMs.

In contrast to use case steps, tasks do not have an alternative outcome and the execution of a task has only one result. Figure 8 depicts the FSM denoting an action task. It consists of only one initial and one final state. The transition between the two states is triggered by an event denoting the completion of task execution.

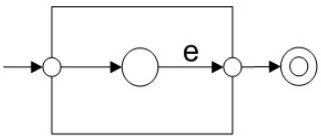


Fig. 8. FSM Representing an Action Task

In what follows we demonstrate how CTT temporal operators (using the example of *enabling* (>>) and *choice* ([])) are semantically mapped into compositions of FSMs. The sequential execution of two tasks (denoted by the *enabling* operator) is semantically mapped into the sequential composition of the corresponding state machines. As each FSM representing a task has only one final state, the sequential composition of two FSMs M_1 and M_2 is performed by simply defining a new lambda transition from the final state of M_1 to the initial state of M_2.

The mapping of the CTT *choice* operator is less trivial. At this point it is important to recall our assumption (see Section 2.3) that task models specify system behavior as an input-output relation, where internal system events are omitted. Moreover the execution of a task can result only in one state. The specification of alternative outcomes is not possible. Both observations have implications on the semantic mapping of the *choice* operator. Depending on the task types of the operands we propose distinguishing between deterministic choices and non-deterministic choices. If the enabled tasks of both operands are application tasks (e.g. "Display Success Message", "Display Failure Message", etc.) then (a) the non-deterministic choice is used to compose the corresponding FSMs, otherwise (b) the deterministic choice composition is employed.

The former (a) is justified by the fact that each application works in a deterministic manner. Hence, the reason why the system performs either one task or the other is because the internal states of the system are not the same. Depending on its internal state, the system either performs the task specified by the first operand or the task specified by the second operand. However, task models do not capture internal system operations. As a result, from the task model specification, we do not know why the system is in one state or the other and the choice between the states becomes non-deterministic.

As for the latter case (b), the choice (e.g. between two interaction tasks) is interpreted as follows. *In a given state of the system, the user has the exclusive choice between carrying one or the other task.* Clearly the system may only be in one possible state when the choice is made. Hence, the deterministic choice composition is applicable.

Table 1 schematically depicts the difference between deterministic choice composition and non-deterministic choice composition of two FSMs. In contrast to deterministic choice composition (discussed in the previous section) non-deterministic choice composition does not merge the initial states of the involved FSMs, but introduces a new initial state.

Table 1: Choice Compositions of FSMs

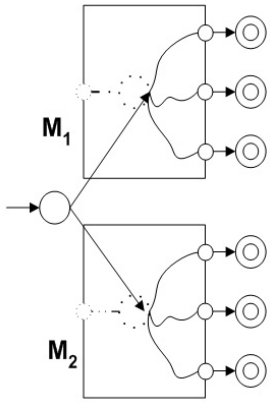
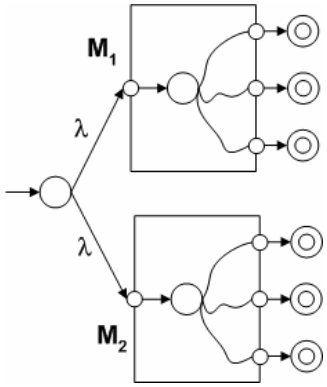
Deterministic Choice Composition	**Non-deterministic Choice Composition**
	

Figure 9 portrays the corresponding FSM for the "Login" task model. We note that the non-deterministic choice composition has been employed to denote the CTT choice between the system tasks "Display Success Message" and "Display Failure Message". After the execution of the "Submit" task the system non-deterministically results in two different states. Depending on the state either the Failure or the Success Message is displayed.

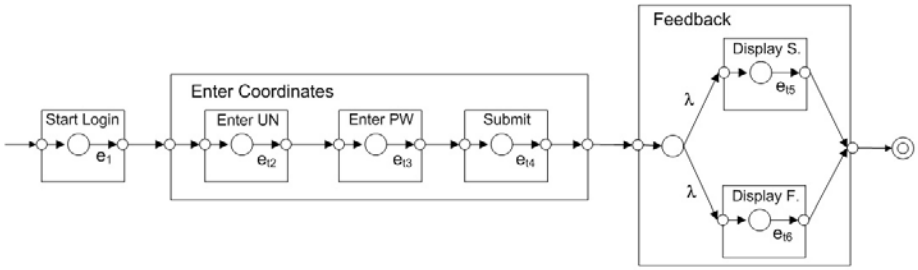


Fig. 9. FSM Representation of the "Login" Task Model

For the sake of completeness we now briefly sketch out how the remaining CTT operators (besides *enabling* and *choice*) can be mapped into FSM compositions: In CTT it is possible to declare tasks as *iterative* or *optional*. Iterative behavior can be implemented by adding a transition from the final state to the initial state of the FSM representing the task, whereas optional behavior may be implemented by adding a lambda transition from the initial state to the final state. The remaining CTT operators are more or less a short hand notation for more complex operations. As such they can be rewritten using the standard operators. For example the *order independency* (t_1 |-| t_2) operator can be rewritten as the choice of either executing t_1 followed by t_2 or executing t_2 followed by t_1. Another example is the *concurrency* (t_1 ||| t_2) operator, which can be rewritten as the choice between all possible interleavings of action tasks entailed in t_1 and t_2. Similar rewritings can be established for the operators *disabling* and *suspend/resume*. Further details can be found in [18].

3.5 A Formal Definition of Consistency

In Section 2.3 we made the assumption and viewed task models as UI specific implementations of a use case specification. In this section we will tackle the question of what is the class of valid task model implementations for a given use case specification. To this effect we propose the following two consistency principles:

1. Every scenario in the task model is also a valid scenario in the use case specification. That is, *what the implementation* (task model) *does is allowed by the specification* (use case).
2. Task models do not capture internal operations, which are however specified in the corresponding use case specification. In order to compensate for this allowed degree of under-specification we require the task model to *cater for all possibilities that happen non-deterministically* from the user's perspective.

For example as specified by the "Login" use case the system notifies the primary actor of the success or failure of his login request based on the outcome of the *internal* validation step. According to the second consistency principle we require every task model that implements the "Login" use case specification to specify the choice between a task representing the success notification and a task representing the failure notification.

We note that the first consistency principle can be seen as a safety requirement, as it enforces that *nothing bad can happen* (the task model must not specify an invalid scenario with respect to the use case specification). The second consistency principle can be seen as a liveness requirement as it ensures that the task model specification does not "deadlock" due to an unforeseen system response.

In order to formalize the two consistency principles we adopt Brinksma's extension relation [15], which tackles a related conformance problem for labeled transition systems. Informally, a use case specification and a task model specification are consistent, if and only if the later is an extension of the former. Our definition of consistency between task models and use cases is as follows:

Definition 7: Consistency. Let $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ be the FSM representing the use case U and $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be the FSM representing the task model T. Then T is consistent to the use case U iff the following two properties hold.

(1) **Language inclusion** (safety property)

$L(M_2) \subseteq L(M_1)$

(2) **Sufficient coverage:** (liveness property)

$\forall t \in T$ with $T = \{Traces(M_2) \setminus L(M_2)\}$

a. Let $Q_{M1}=\{p_1, p_2, \ldots, p_n\}$ be $\delta^*(q_{01},t)$. That is, the p_i's are all and only the states that can be reached from the initial state of M_1 after having accepted t.

b. Let $Q_{M2}=\{q_1, q_2, \ldots, q_m\}$ be $\delta^*(q_{02},t)$. That is, the q_j's are all and only the states that can be reached from the initial state of M_2 after having accepted t.

c. **We require that:** $\forall p \in Q_{M1} \exists q \in Q_{M2}$. failure (p) $\subseteq$ failure (q).

The liveness property states that the task model FSM must refuse to accept an event in a situation where the use case FSM may also refuse. If we translate this condition back to the domain of use cases and task models, we demand the task model to provide a task for every situation where the use case must execute a corresponding step. The main difference to Brinksma's original definition is that our definition is defined over finite state machines instead of labeled transition systems. As a consequence, we require that the language accepted by the task model FSM is included in the language accepted by the use case FSM (safety property). Task models that only implement partial scenarios of the use case specification are deemed inconsistent.

One precondition for the application of the definition is that both state machines operate over the same alphabet. The mappings described in the previous sections do not guarantee this property. Hence, in order to make the FSMs comparable, a set of preliminary steps have to be performed and are described in the following:

1. **Abstraction from internal events:** Task models do not implement internal system events. Hence, we require the alphabet of the use case FSM to be free of symbols denoting internal events. This can be achieved by substituting every symbol denoting an internal event by lambda (λ)[2].
2. **Adaptation of abstraction level:** Task model specifications are (typically) at a lower level of abstraction than their use case counter parts. As such a use case step may be refined by several tasks in the task model. Events representing the execution of these refining tasks will hence not be present in the use case FSM. We therefore require that for every event 'e' of the task model FSM there exists a bijection that relates 'e' to one corresponding event in the use case FSM. This can be achieved by replacing intermediate lower level events in the task model FSM with lambda events. Events denoting the completion of a refining task group are kept.
3. **Symbol mapping:** Finally, the alphabets of the two FSMs are unified by renaming the events of the task model FSM to their corresponding counterparts in the use case FSM.

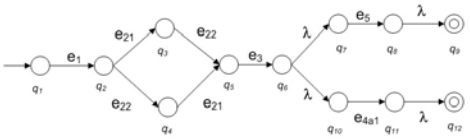
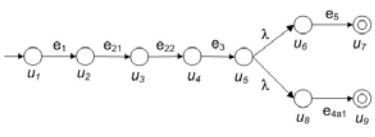
In what follows we will apply our consistency definition to verify that the "Login" task model is a valid implementation of the "Login" use case. Table 2 depicts the FSMs for the "Login" use case (M_U) and the "Login" task model (M_T), after the unification of their input alphabets. We start with the verification of the safety property (language inclusion). With

$L(M_U)=\{<e_1,e_{21},e_{22},e_3,e_5>,<e_1,e_{22},e_{21},e_3,e_5>,<e_1,e_{21},e_{22},e_3,e_{4a1}>,<e_1,e_{22},e_{21},e_3,e_{4a1}>\}$

$L(M_T)=\{<e_1, e_{21}, e_{22}, e_3, e_5>,<e_1, e_{21}, e_{22}, e_3, e_{4a1}>\}$

we can easily see the $L(M_T) \subseteq L(M_U)$. Hence the first property is fulfilled.

Table 2: Use Case FSM and Task Model FSM After the Unification of Their Alphabets

Unified Use Case FSM (M_U)	**Unified Task Model FSM (M_T)**
	

We continue with the verification of the second property (liveness). The set T of all partial runs of M_T is as follows:

$T = \{<e_1>,<e_1,e_{21}>,<e_1,e_{21},e_{22}>, <e_1,e_{21},e_{22},e_3>\}$

We verify for each trace t in T that the liveness property holds. Starting with $t= <e_1>$ we obtain $Q_{MU}=\{q_2\}$; $Q_{MT}=\{u_2\}$ as the set of reachable states in M_U and M_T after having accepted t. Next we verify that for every state in Q_{MU} there exists a state in Q_{MT} with an encompassing failure set. Since Q_{MU} and Q_{MT} only contain one element we require that failure $(q_2) \subseteq$ failure (u_2). With failure$(q_2) = \{e_1, e_3, e_5, a_{4a1}\}$ and failure$(u_2) = \{e_1, e_{22}, e_3, e_5, a_{4a1}\}$ this property is clearly fulfilled. In a similar fashion

[2] Lambda denotes the empty string and as such is not part of the language accepted by an FSM.

we prove that the liveness property holds for the traces: <e_1,e_{21}>,<e_1,e_{21},e_{22}>. More interesting is the case where t = <e_1,e_{21},e_{22},e_3>. We obtain Q_{MU}={q_6, q_7, q_{10}}; Q_{MT}={u_5, u_6, u_8} as the set of reachable states in M_U and M_T after having accepted t. Next we have to find for each state in Q_{MU} a state in Q_{MT} with an "encompassing" failure set. For q_6 (failure(q_6)={e_1, e_{21}, e_{22}, e_3}) we identify u_5 (failure(u_5)={e_1, e_{21}, e_{22}, e_3}). For q_7 (failure(q_7)={e_1, e_{21}, e_{22}, e_3, e_{4a1}}) we identify u_6 (failure(u_6)={e_1, e_{21}, e_{22}, e_3, e_{4a1}}) and for q_{10} (failure (q_{10})= {e_1, e_{21}, e_{22}, e_3, e_5}) we identify u_8 (failure (u_8) = {e_1, e_{21}, e_{22}, e_3, e_5}). For each identified pair of p_i and q_i it can be easily seen that failure (p_j) $\subseteq$ failure (q_i), hence we conclude that the "Login" task model represented by M_T is consistent to the "Login" use case represented by M_U q.e.d.

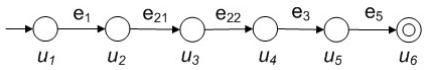


Fig. 10. FSM Representation of an Inconsistent "Login" Task Model

We conclude this chapter with a counter example, by presenting a "Login" task model which is not a valid implementation of the "Login" use case. The FSM (M_{T2}) portrayed by Figure 10 represents a task model which does not contain the choice between "Display Failure Message" and "Display Success Message". Instead, after the "Submit" task (e3), "Success Message" (e5) is always displayed. It can be easily seen that the safety property holds with $L(M_{T2}) \subseteq L(M_U)$. The verification of the liveness property however will lead to a contradiction. For this purpose, let us consider the following trace of M_{T2}: t = <e_1,e_{21},e_{22},e_3>. We obtain Q_{MU}={q_6, q_7, q_{10}} and Q_{MT2}={u_5} as the set of all reachable states in M_U and M_T after having accepted t. In this case however, for q_{10} we cannot find a corresponding state in Q_{MT2} (which in this case consists of a single element only) such that the failure set inclusion holds. We obtain failure(q_{10})={e_1, e_{21}, e_{22}, e_3, e_5} and failure(u_5)={e_1, e_{21}, e_{22}, e_3, e_{4a1}}. Clearly failure(q_{10}) is not a subset of failure(u_5). Hence the task model is not consistent to the "Login" use case.

4 Conclusion

In this paper we proposed a formal definition of consistency between use cases and task models based on a common formal semantics. The main motivation for our research is the need for an integrated development methodology where task models are developed as logical progressions from use case specifications. This methodology rests upon a common semantic framework where we can formally validate whether a task model is consistent with a given use case specification. With respect to the definition of the semantic framework, we reviewed and contrasted key characteristics of use cases and task models. As a result we established that task model specifications are at a lower level of abstraction than their use case counterparts. We also noted that task models omit the specification of internal system behavior, which is present in use cases.

These observations have been used as guides for both the mapping to finite state machines and for the formal definition of consistency. The mapping is defined in a compositional manner over the structure of use cases and task models. As for the definition of consistency, we used an adaptation of Brinksma's extension pre-order. We found the extension relation appropriate because it acknowledges the fact that under certain conditions two specifications remain consistent, even if one entails additional behavioral information which is omitted in the second. Both the mapping and the application of the proposed definition of consistency have been supported by an illustrative example.

As future work, we will be tackling the question of how relationships defined among use cases (i.e. *extends* and *includes)* can be semantically mapped into finite state machines. This will allow us to apply the definition of consistency in a broader context, which is not restricted to a single use case. Another issue deals with the definition of consistency among two use case specifications and in this vein also among two task model specifications. For example, if a user-goal level use case is further refined by a set of sub-function use cases it is important to verify that the sub-function use cases do not contradict the specification of the user goal use case. Finally we note that for the simple "Login" example consistency can be verified manually. However, as the specifications become more complex, efficient consistency verification requires supporting tools. We are currently investigating how our approach can be translated into the specification languages of existing model checkers and theorem provers.

Acknowledgements

This work was funded in part by the National Sciences and Engineering Research Council of Canada. We are grateful to Homa Javahery who meticulously reviewed and revised our work.

References

1. Seffah, A., M. C. Desmarais and M. Metzger, Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues, chapter in *Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*, Springer.
2. Cockburn, A., Writing effective use cases, Addison-Wesley, Boston, 2001.
3. Pressman, R. S., Software engineering: a practitioner's approach, McGraw-Hill, Boston, Mass., 2005.
4. Larman, C., Applying UML and patterns : an introduction to object-oriented analysis and design and the unified process, Prentice Hall PTR, Upper Saddle River, NJ, 2002.
5. XSLT, XSL Transformations Version 2.0 [Internet], Available from http://www.w3.org/TR/xslt20/ , Accessed: Dec. 2006, Last Update: Nov. 2006.
6. Paternò, F., Towards a UML for Interactive Systems, in *Proceedings of EHCI 2001*, Toronto, Canada, pp. 7-18, 2001.

7. Souchon, N., Q. Limbourg and J. Vanderdonckt, Task Modelling in Multiple contexts of Use, in *Proceedings of Design, Specification and Verification of Interactive Systems*, Rostock, Germany, pp. 59-73, 2002.
8. Mori, G., F. Paternò and C. Santoro, CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, in IEEE Transactions on Software Engineering, August 2002, pp. 797-813, 2002.
9. Paternò, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.
10. Bowman, H., Steen, M. W. A., Boiten, E. A., Derrick, J., A Formal Framework for Viewpoint Consistency, *Formal Methods in System Design*, p.111-166, September 2002.
11. Ichikawa, H., Yamanaka, K., and J. Kato, "Incremental specification in LOTOS," in Proc. of *Protocol Specification, Testing and Verification X*, Ottawa, Canada, 1990, pp. 183–196.
12. De Nicola, R., Extensional Equivalences for Transition Systems, Acta Informatica, Vol. 24, pp. 211-237, 1987.
13. Butler, M. J., A CSP Approach to Action Systems, PhD Thesis in Computing Laboratory, Oxford University, 1992.
14. Khendek, F., Bourduas, S., Vincent, D., Stepwise Design with Message Sequence Charts, Proceedings *of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Cheju Island, Korea, August 28-31 2001.
15. Brinksma, E., Scollo, G., Steenbergen, C., LOTOS specifications, their implementations, and their tests, in Proceedings of IFIP Workshop Protocol Specification, Testing, and Verification VI, pp. 349-360, 1987.
16. Bergstra, J. A. 2001 *Handbook of Process Algebra*. Elsevier Science Inc.
17. Brookes, S. D., Hoare, C. A. R., Roscoe, A. D., A Theory of Communicating Sequential Processes, Journal of ACM, Vol. 31, No. 3, pp. 560-599, 1984.
18. Sinnig, D., Chalin, P., Khendek, F., Towards a Common Semantic Foundation for Use Cases and Task Models, to Appear in *Electronic Notes in Theoretical Computer Science (ENTCS)*, Dec. 2006.
19. Paternò F., Santoro C., The ConcurTaskTrees Notation for Task Modelling, Technical Report at CNUCE-C.N.R., May, 2001.
20. Xu, J., W. Yu, K. Rui and G. Butler, Use Case Refactoring: A Tool and a Case Study, in Proceedings of APSEC 2004, Busan, Korea, pp. 484-491, 2004.
21. Kosters, G. Pagel, B., Winter, M., Coupling Use Cases and Class Models, in *Proceedings of the BCS-FACS/EROS workshop on "Making Object Oriented Methods More Rigorous", Imperial College*, London, June 24th, 1997, pp. 27-30.
22. Mizouni, R., A. Salah, R. Dssouli and B. Parreaux, Integrating Scenarios with Explicit Loops, in *Proceedings of NOTERE*, 2004, Essaidia Morocco, 2004.
23. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.-M., Automatic test generation: a use case driven approach, IEEE Transactions on Software Engineering, Volume 32, Issue 3, March 2006, pp. 140 - 155.
24. Hopcroft, J. E., Motwani, R., Ullman, J. D., Introduction to Automata Theory, Languages, and Computation (3rd Edition), Addison Wesley, 2006.